Trabalho Prático 2 - Sistemas Operacionais

Jean G. A. Evangelista
2018047021
Luiz A. D. Berto
2018047099

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

jeanalves@dcc.ufmg.br

luizberto@dcc.ufmg.br

Abstract. *This work aims to modify the preemption and process scheduling policies from the XV6 operating system. The original round-robin algorithm was replaced in favor an implementation of the multi-level queue scheduling algorithm, and empirical analysis showed that for the set of tests proposed, the MLMQ algorithm disfavors S-bound (CPU-bound with short bursts) processes, and favors IO-bound processes.*

Resumo. *Esse trabalho visa modificar as políticas de preempção e escalonamento de processos do sistema operacional XV6. O algoritmo original round-robin foi substituido por uma implementação do algoritmo de filas multinível, e uma análise empírica mostrou que para o conjunto de testes proposto, o algoritmo de filas multinível desfavorece os processos S-bound (CPU-bound com rajões curtos de processamento), e favorece os processos IO-bound.*

1. Introdução

O XV6 é um sistema operacional bastante simples [xv6 2020] e muito utilizado para ensino. Por meio dele é possível visualizar e trabalhar na prática com inúmeros conceitos estudados na área de Sistemas Operacionais. Nesse trabalho foram abordados tópicos como preempção, escalonamento de processos, chamadas de sistema, entre outros.

2. Escalonador de Processos

2.1. Modificação no processo de preempção

A primeira tarefa realizada consiste na modificação da política do processo de preempção do XV6. Conforme especificado, era necessário que o processo de preempção ocorresse a cada *n* unidades de tempo (ticks) e não a cada 1 unidade de tempo. Para realizar isso, foi necessário definir uma constante *INTERV* no arquivo *param.h* e utilizá-la numa chamada do método *lapicw*, multiplicando-a pelo 1 tick já definido.

2.2. Alteração da prioridade de um processo

Foi criada a chamada de sistema *set_prio* no arquivo *proc.c* para realizar a alteração na prioridade um processo. A chamada recebe como parâmetro um número inteiro no intervalo [0,2] que representa a nova prioridade do processo. Após validar a nova prioridade recebida, o método obtém o lock da tabela de processos, altera a prioridade do processo atual e libera o lock.

2.3. Modificação da política de escalonamento de processos

A política de escalonamento de processos foi modificada, passando utilizar o algoritmo de Escalonamento de Filas Multinível. Foi necessário implementar uma estrutura de fila no arquivo *proc.c*. Foram criadas 3 filas, de capacidade máxima *NPROC* (número máximo de processos), para cada prioridade. No método *allocproc* a prioridade inicial do processo é definida como 2, e ele é enfileirado na fila referente aos processos com essa prioridade. A chamada de sistema *set_prio* implementada anteriormente também foi alterada: antes de realizar a alteração da prioridade do processo, a prioridade atual é obtida e o processo é remanejado para a fila correta. Por último, o método *scheduler* foi modificado para obter o processo a ser executado de uma das três filas criadas, respeitando sempre a ordem de prioridade 2, 1, 0: o *scheduler* só procura um processo na fila 1 caso nenhum processo tenha sido encontrado na fila 2, e o mesmo vale para a fila 0 em relação à fila 1. Uma vez obtido o processo, ele é movido para o fim da sua fila de prioridade.

2.4. Aging

Foi necessário implementar um mecanismo de *aging* para evitar a ocorrência de inanição. Primeiramente foram adicionadas as constantes *ZERO_TO_ONE* e *ONE_TO_TWO*, que guardam a informação do tempo de espera de um processo. Em seguida, foram adicionados os membros *ctime*, *stime*, *retime* e *rutime* na estrutura *proc*, que também foram utilizados nos testes. A implementação do mecanismo de *aging* é bastante simples: caso a prioridade do processo for 0 e seu *retime* for maior ou igual a *ZERO_TO_ONE*, sua prioridade passará de 0 para 1; similarmente, caso a prioridade do processo for 1 e seu *retime* for maior ou igual a *ONE_TO_TWO*, sua prioridade passará de 1 para 2. O processo de verificação e atualização descrito anteriormente é realizado a cada *tick* de *clock*.

3. Testes

Conforme especificado, era necessário realizar uma análise do impacto das alterações realizadas. Como já citado, foram adicionados os campos *ctime*, *stime*, *retime* e *rutime* na estrutura do *proc*. Tais campos guardam o tempo em que o processo foi criado, o tempo no estado *SLEEPING*, o tempo no estado *RUNNABLE* e o tempo no estado *RUNNING*, respectivamente.

3.1. Extração das informações de um processo

Para extrair as informações de um processo foi criada a chamada de sistema *wait2*. O que o *wait2* faz é basicamente o que o *wait* faz: percorre toda a tabela de processos até achar um processo filho "morto" (estado igual a *ZOMBIE*). Uma vez encontrado, as estatísticas do processo filho são armazenadas e o PCB deste é desalocado, e o *wait2* encerra sua execução. Caso nenhum filho seja encontrado ou o processo atual esteja morto é retornado -1.

3.2. Programa de Testes

Por último, foi criado o programa *sanity.c* para efetivamente realizar os testes. O programa recebe um inteiro *n* como parâmetro, que define quantos processos serão criados ($5n$). O programa cria $5n$ processos, que são alocados em três categorias:

1. Processos CPU-bound, que executam a instrução *nop* (invocada diretamente via asm) 100*1000000 vezes;

2. Processos S-bound, que são similares aos processos CPU-bound, mas eles entregam a CPU (via yield) a cada 10000 iterações;
3. Processos IO-bound, que chamam *sleep(1)* 100 vezes.

Para cada processo, o programa *sanity* chama a syscall *wait2*, e imprime em *stdout* o PID do processo que morreu, a qual categoria ele faz parte (CPU-bound, S-bound ou IO-bound), e as estatísticas de execução deste (*rutime*, *retime* e *stime*). Após isso, o programa agrega as estatísticas retornadas por *wait* nos arrays de estatísticas agregadas, sempre na posição *PID % 3*.

Após todos os processos terminarem, o programa calcula as médias das estatísticas, dividindo a soma de cada uma pela quantidade de processos de cada tipo, e imprime em tela um relatório da execução, informando o tipo, a quantidade de processos, e o tempo médio dos processos daquele tipo nos estados *SLEEPING* e *RUNNING* e o tempo médio de *turnaround* (note que o tempo de turnaround é simplesmente *rutime* + *retime* + *stime*).

3.3. Resultados

Com base nos resultados obtidos, é possível afirmar que processos do tipo S-bound são prejudicados no algoritmo de escalonamento de filas multinível. Após executar o programa *sanity* com valores de *n* variando de 1 a 12, pôde-se perceber que os processos do tipo S-bound tinham tempos de espera muito mais altos que os processos de outras categorias. Foi possível perceber uma relação linear entre a média de *retime* e n para todos os tipos de processos, mas para os processos do tipo S-bound a constante é muito maior. Isso acontece pois esses processos executam por pouco tempo e liberam o processador, e como o algoritmo coloca o processo no final da fila ao selecioná-lo para execução, o tempo até que ele seja selecionado novamente pode ser alto.

Por outro lado, pôde-se perceber também que os processos IO-bound foram beneficiados, tendo um *retime* bastante inferior ao que seria observado no algoritmo de escalonamento original do XV6, por exemplo: no algoritmo *round-robin*, no pior caso o processo IO-bound poderia ser o último da fila logo após sair da espera de IO; já no algoritmo de filas multinível, os processos IO-bound tendem a ficar no começo da fila, visto que os processos que são executados vão sendo jogados para o final da fila, o que efetivamente minimiza *retime*.

3.4. Gráficos

Apresentamos alguns gráficos que mostram as estatísticas de cada tipo de processo em execuções de *sanity*, variando *n* de 1 a 12.

3.4.1. Estatísticas de processos CPU-bound

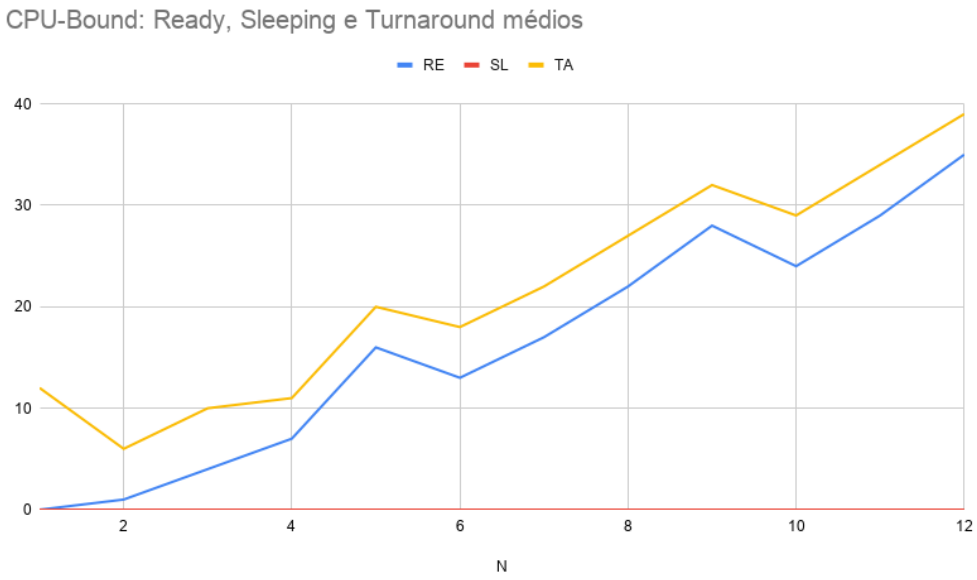

Figure 1. CPU-Bound: Ready, Sleeping e Turnaround médios

Podemos perceber que o *retime* tem crescimento linear com constante baixa para processos CPU-bound.

3.4.2. Estatísticas de processos S-bound

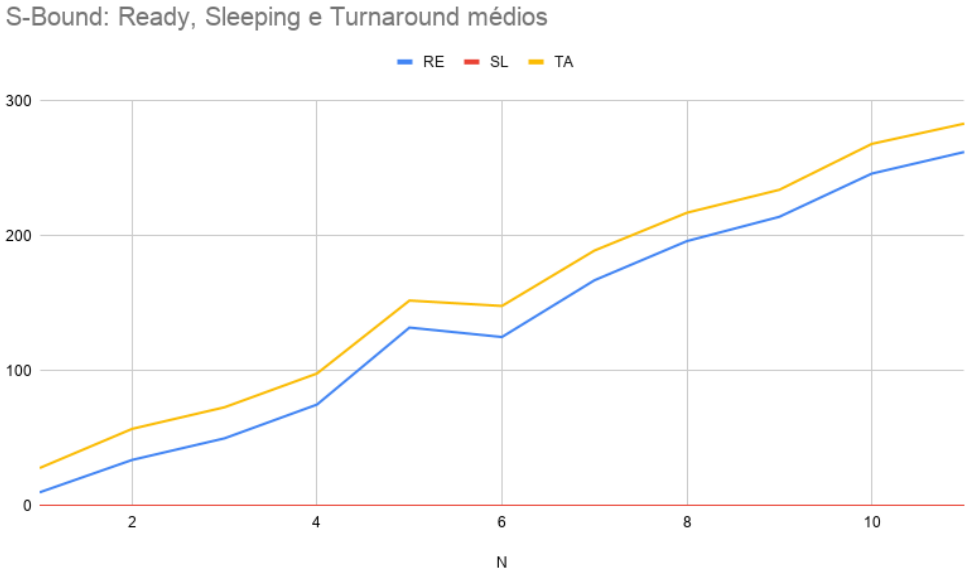

Figure 2. S-Bound: Ready, Sleeping e Turnaround médios

Diferentemente dos processos CPU-bound, a constante que multiplica o fator linear de crescimento do *retime* para os processos S-bound é alta.

3.4.3. Estatísticas de processos IO-bound

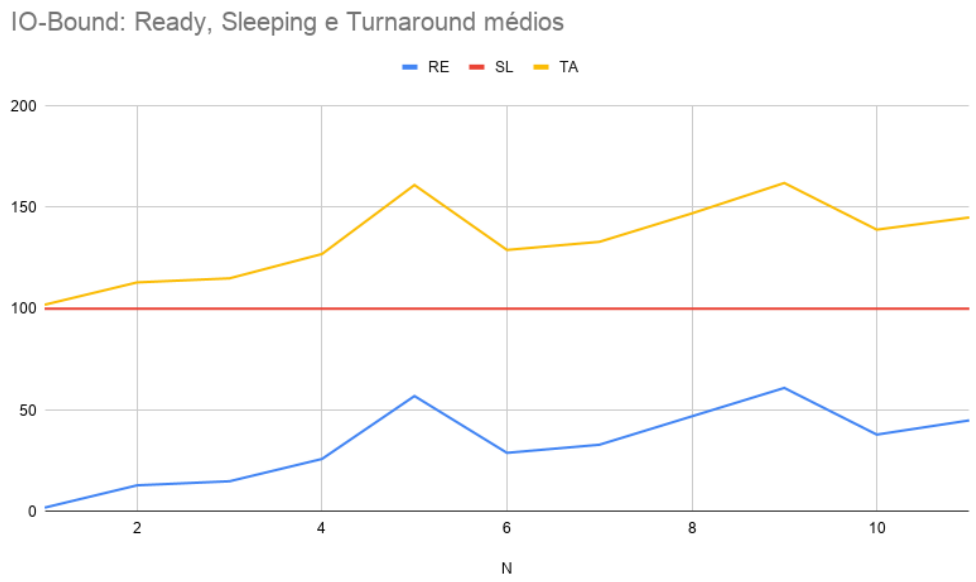

Figure 3. IO-Bound: Ready, Sleeping e Turnaround médios

Assim como para os processos CPU-bound, os processos IO-bound tem crescimento linear com constante baixa.

4. Conclusão

Primeiramente é necessário observar que o XV6 foi uma ferramenta excelente para praticar muitos dos conceitos vistos nas aulas, como chamadas de sistema e escalonamento. Também é importante observar que com poucas modificações foi possível alterar de maneira significativa as políticas do sistema operacional.

Por fim, podemos concluir que algoritmos de escalonamento são complexos, de performance volátil, e no geral são de difícil classificação determinística, visto que a performance deles depende de qual métrica está sendo avaliada e também do estado do sistema operacional (no que diz respeito à quantidade e tipo de processos em atividade).

References

(2020). Xv6 source code. `https://github.com/mit-pdos/xv6-public`. Acessado em: 2020-19-09.